**What is Python?**

In this introductory section, we'll understand what Python is and how we can set it up.

**We'll cover the following**

* + [The nature of the language](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/Y5wJrN9Pn2K#The-nature-of-the-language)
    - [A High-Level Language](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/Y5wJrN9Pn2K#A-High-Level-Language-)
    - [Readability](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/Y5wJrN9Pn2K#Readability-)
  + [Applications](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/Y5wJrN9Pn2K#Applications-)
  + [Version History](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/Y5wJrN9Pn2K#Version-History)

Welcome to this module! We’re glad to have you with us on this journey through the realm of **Python**.

Since we’ll be starting from scratch, there’s no need to worry if you have no prior experience with Python or coding in general.

So, without further ado, let’s get started by learning what Python is.

**The nature of the language**[#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/Y5wJrN9Pn2K#The-nature-of-the-language)

Developed in 1990, Python is one of the most popular *general-purpose* programming languages in modern times.

The term “general-purpose” simply means that Python can be used for a variety of applications and does not focus on any one aspect of programming.

**A High-Level Language**[#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/Y5wJrN9Pn2K#A-High-Level-Language-)

Python falls under the category of **high-level, interpreted** languages. A high-level language is one which cannot be understood directly by our machine. There is a certain degree of *abstraction* in its syntax. Machines are generally designed to read **machine code**, but high-level syntax cannot be directly converted to machine code.

As a result, it must first be converted to **bytecode** which is then converted to machine code before the program can be executed.

Python is an interpreted language because, during execution, each line is interpreted to the machine language on-the-go.

However, if we take the example of C++, the code needs to be compiled into an executable first, and then it can be executed. In Python, we can skip this compilation step (Python does it for us behind the scenes) and directly run the code.

**Readability**[#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/Y5wJrN9Pn2K#Readability-)

One of the biggest reasons for Python’s rapid growth is the simplicity of its syntax. The language reads almost like plain English, making it easy to write complex programs.

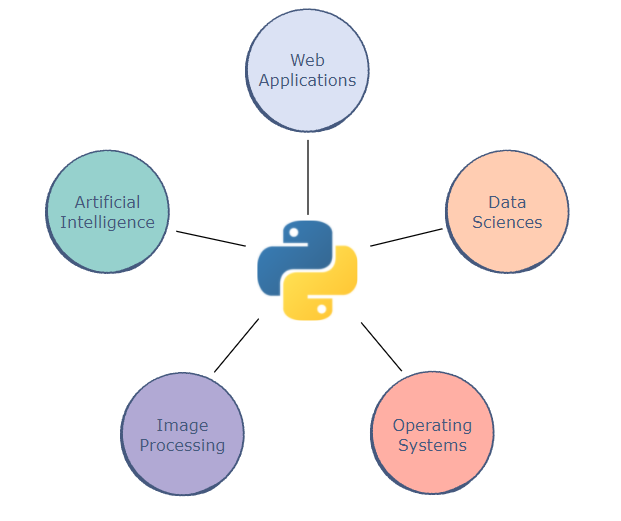
Since it doesn’t have much of a learning curve, Python is a very good entry point into the world of programming for beginners.

**Applications**[#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/Y5wJrN9Pn2K#Applications-)

Apart from the ease of learning, Python is a very efficient language which is used in almost every sphere of modern computing.

This makes a strong case for learning Python, even for non-programmers.

Some of Python’s main applications are highlighted below:



**Version History**[#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/Y5wJrN9Pn2K#Version-History)

Python has had several major updates in the past. **Python 2.7** was widely used for a very long time, even after the release of newer versions.

However, Python 2.7 has been deprecated as of January 01 2020, and replaced completely by 3.xx versions, known as **Python 3**. The differences between Python 2.7 and Python 3 are minute, but important nonetheless.

To keep up with the latest technologies, we’ll be dealing with Python 3 for the entirety of this module.

Throughout this module, we will be able to write and execute Python code right here 🎉.

**Writing Our First Code**

In this lesson, we'll examine one of the simplest codes in Python syntax.

**We'll cover the following**

* + [The print Statement](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/R8myL3DYY9R#The-print-Statement-)
    - [Printing Multiple Pieces of Data](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/R8myL3DYY9R#Printing-Multiple-Pieces-of-Data-)
  + [Comments](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/R8myL3DYY9R#Comments-)

By now, we’ve learned what kind of language Python is. We are finally ready to start writing code! So, let’s move on to the fun stuff.

**The print Statement**[#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/R8myL3DYY9R#The-print-Statement-)

Whenever we learn a new language, it is an age-old tradition to start by displaying the text “**Hello World**” on the screen. For the remainder of this course, the terminal will act as our screen.

Every language has a different syntax for displaying or *printing* something on the screen.

Since Python is one of the most readable languages out there, we can print data on the terminal by simply using the print statement.

Here’s what the statement looks like:

print (data)

Whatever we need to print is encapsulated in the parentheses following the print keyword. Let’s try printing “Hello World” on the terminal:

print(50)

print(1000)

print(3.142)

1

2

print("Hello World")

The text Hello World is bounded by quotation marks because it is a *string* or a group of characters, more on this later.

Next, we’ll print a few numbers. Each call to print moves the output to a new line:

print(50)

print(1000)

print(3.142)

**Printing Multiple Pieces of Data**[#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/R8myL3DYY9R#Printing-Multiple-Pieces-of-Data-)

We can even print multiple things in a single print command; we just have to separate them using **commas**.

By default, each print statement prints text in a new line. If we want multiple print statements to print in the same line, we can use the following code:

print("Hello", end="")

print("World")

print("Hello",·end="·")

print("World")

Output

2.55s

HelloWorld

Hello World

The value of end is appended to the output and the next print will continue from here.

**Comments**[#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/R8myL3DYY9R#Comments-)

**Comments** are pieces of text used to describe what is happening in the code. They have no effect on the code whatsoever.

A comment can be written using the # character:

print(50)  # This line prints 50

print("Hello World")  # This line prints Hello World

# This is just a comment hanging out on its own!

# For multi-line comments, we must

# add the hashtag symbol

# each time

An alternative to these multi-line comments (line 4 - 8) are **docstrings**. They are encased in triple quotes, """, and can be used to replace multi-line comments:

""" Docstrings are pretty cool

for writing longer comments

or notes about the code"""

That brings us to the end of this section. Be sure to check out the quiz in order to test what you have learned so far.

In the next section, we will learn about the different data types and operators in Python.

**What are Data Types and Variables?**

In this section, we'll learn about the different data types in Python.

**We'll cover the following**

* + [Definition](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/xlwqJM9l8mB#Definition)
  + [Python’s Data Types](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/xlwqJM9l8mB#Python%E2%80%99s-Data-Types)
  + [Variables](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/xlwqJM9l8mB#Variables)
  + [Naming Convention](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/xlwqJM9l8mB#Naming-Convention)

**Definition**[#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/xlwqJM9l8mB#Definition)

The **data type** of an item defines the type and range of values that item can have.

The concept of data types can be found in the real world. There are numbers, alphabets, characters, etc., that all have unique properties due to their classification.

Such a classification is also made in many programming languages, including Python.

**Python’s Data Types**[#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/xlwqJM9l8mB#Python%E2%80%99s-Data-Types)

Unlike many other languages, Python does not place a strong emphasis on defining the data type of an object, which makes coding much simpler. The language provides three main data types:

* **Numbers**
* **Strings**
* **Booleans**

**Variables**[#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/xlwqJM9l8mB#Variables)

A variable is simply a name to which a value can be *assigned*.

Variables allow us to give meaningful names to data.

The simplest way to assign a value to a variable is through the = operator.

A big advantage of variables is that they allow us to store data so that we can use it later to perform operations in the code.

Variables are **mutable**. Hence, the value of a variable can always be updated or replaced.

**Naming Convention**[#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/xlwqJM9l8mB#Naming-Convention)

There are certain rules we have to follow when picking the name for a variable:

* The name can start with an upper or lower case alphabet.

For example, you can define your income variable as Income or income, both are valid.

* All the names are case sensitive.

For example, Income and income are two different variables and not one.

* A number can appear in the name, but not at the beginning.

For example, 12income is not a valid name but income12 or in12come are valid.

* The \_ character can appear anywhere in the name.

For example, \_income or income\_ are valid names.

* Spaces are not allowed. Instead, we must use [snake\_case](https://en.wikipedia.org/wiki/Snake_case" \t "_blank) to make variable names readable.

For example, monthly\_income is a valid name.

* The name of the variable should be something meaningful that describes the value it holds, instead of being random characters.

For example, inc or even income would not give any useful information but names like weekly\_income, monthly\_income, or annual\_income explain the purpose of our defined variable

# Numbers

This lesson provides an in-depth discussion about numbers in Python.

Python is one of the most powerful languages when it comes to manipulating numerical data.

It is equipped with support for several types of numbers, along with utilities for performing computations on them.

There are three main types of numbers in Python:

**Integers**

The integer data type is comprised of all the positive and negative whole numbers.

The amount of memory an integer occupies depends on its value. For example, 0 will take up 24 [*bytes*](https://en.wikipedia.org/wiki/Byte) whereas 1 would occupy 28 bytes.

Here are some examples of integers:

8

print(10)  # A positive integer

print(-3000)  # A negative integer

num = 123456789  # Assigning an integer to a variable

print(num)

num = -16000  # Assigning a new integer

print(num)

Output

1.82s

10

-3000

123456789

-16000

**Note**: In Python, all negative numbers start with the - symbol.

Floating-point numbers in Python.

**Floating Point Numbers**

Floating-point numbers, or **floats**, refer to positive and negative decimal numbers.

Python allows us to create decimals up to a very high decimal place.

This ensures accurate computations for precise values.

A float occupies 24 bytes of memory.

Below, we can find some examples of floats:

print(1.00000000005)  # A positive float

print(-85.6701)  # A negative float

flt\_pt = 1.23456789

print(flt\_pt)

In Python, 5 is considered to be an integer while 5.0 is a float.

**Complex Numbers**

Python also supports complex numbers, or numbers made up of a real and an imaginary part.

Just like the print() statement is used to print values, complex() is used to create complex numbers.

It requires two values. The first one will be the real part of the complex number, while the second value will be the imaginary part.

Here’s the template for making a complex number:

complex(real, imaginary)

Let’s see a few examples:

Complex numbers in Python.

print(complex(10, 20))  # Represents the complex number (10 + 20j)

print(complex(2.5, -18.2))  # Represents the complex number (2.5 - 18.2j)

complex\_1 = complex(0, 2)

complex\_2 = complex(2, 0)

print(complex\_1)

print(complex\_2)

Output

1.02s

(10+20j)

(2.5-18.2j)

2j

(2+0j)

**Note**: In normal mathematics, the imaginary part of a complex number is denoted by i. However, in the code above, it is denoted by j. This is because Python follows the electrical engineering convention which uses j instead of i. Don’t let that confuse you.

Complex numbers are useful for modelling physics and electrical engineering models in Python. While they may not seem very relevant right now, it never hurts to know!

A complex number usually takes up 32 bytes of memory.

# Booleans

This lesson highlights the key features of the Boolean data type.

The **Boolean** (also known as **bool**) data type allows us to choose between two values: true and false.

In Python, we can simply use True or False to represent a bool:

print(True)

f\_bool = False

print(f\_bool)

Output

0.96s

True

False

**Note**: The first letter of a bool needs to be capitalized in Python.

A Boolean is used to determine whether the logic of an expression or a comparison is correct. It plays a huge role in data comparisons.

**String Slicing**

In this lesson, we'll understand what slicing is and how it can be applied to strings.

**We'll cover the following**

* + [Definition](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/JYJMP8Y2WlK#Definition-)
  + [Slicing with a Step](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/JYJMP8Y2WlK#Slicing-with-a-Step)
  + [Reverse Slicing](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/JYJMP8Y2WlK#Reverse-Slicing-)
  + [Partial Slicing](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/JYJMP8Y2WlK#Partial-Slicing-)

**Definition**[#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/JYJMP8Y2WlK#Definition-)

**Slicing** is the process of obtaining a portion (substring) of a string by using its indices.

Given a string, we can use the following template to slice it and obtain a substring:

string[start:end]

* start is the index from where we want the substring to start.
* end is the index where we want our substring to end.

The character at the end index in the string, will not be included in the substring obtained through this method.

Let’s look at a few examples:

my\_string = "This is MY string!"

print(my\_string[0:4]) # From the start till before the 4th index

print(my\_string[1:7])

print(my\_string[8:len(my\_string)]) # From the 8th index till the end

Output

1.24s

This

his is

MY string!

**Slicing with a Step**[#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/JYJMP8Y2WlK#Slicing-with-a-Step)

Python 3 also allows us to slice a string by defining a **step** through which we can skip characters in the string. The default step is 1, so we iterate through the string one character at a time.

The step is defined after the end index:

string[start:end:step]

Let’s see how this works:

my\_string = "This is MY string!"

print(my\_string[0:7])  # A step of 1

print(my\_string[0:7:2])  # A step of 2

print(my\_string[0:7:5])  # A step of 5

Output

1.22s

This is

Ti s

Ti

**7** of 7

**Reverse Slicing**[#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/JYJMP8Y2WlK#Reverse-Slicing-)

Strings can also be sliced to return a reversed substring. In this case, we would need to switch the order of the start and end indices.

A negative step must also be provided:

my\_string = "This is MY string!"

print(my\_string[13:2:-1]) # Take 1 step back each time

print(my\_string[17:0:2]) # Take 2 steps back. The opposite of what happens in the slide above

Output

1.71s

rts YM si s

!nrsY ish

**Partial Slicing**[#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/JYJMP8Y2WlK#Partial-Slicing-)

One thing to note is that specifying the start and end indices is **optional**.

If start is not provided, the substring will have all the characters until the end index.

If end is not provided, the substring will begin from the start index and go all the way to the end.

Let’s see this in action:

my\_string = "This is MY string!"

print(my\_string[:8])  # All the characters before 'M'

print(my\_string[8:])  # All the characters starting from 'M'

print(my\_string[:])  # The whole string

print(my\_string[::-1])  # The whole string in reverse (step is -1)

Output

20.78s

This is

MY string!

This is MY string!

!gnirts YM si sihT

That’s pretty much all we need to know about string slicing. Play around with the strings above to get a better understanding of how slicing works.

**Operators**

This lesson highlights the different types of operators in Python.

**Operators** are used to perform *arithmetic* and *logical* operations on data. They enable us to manipulate and interpret data to produce useful outputs.

Operators are represented by characters or special keywords.

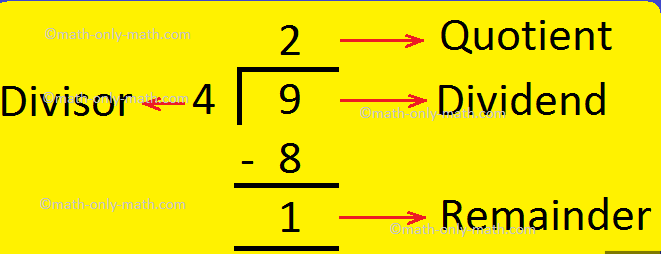
In general, Python’s operators follow the **in-fix** or **prefix** notations.

**In-fix** operators appear between two **operands** (values on which the operator acts) and hence, are usually known as **binary** operators:

A **prefix** operator usually works on one operand and appears before it. Hence, prefix operators are known as **unary** operators:

The 5 main operator types in Python are:

* arithmetic operators
* comparison operators
* assignment operators
* logical operators
* bitwise operators



# Arithmetic Operators

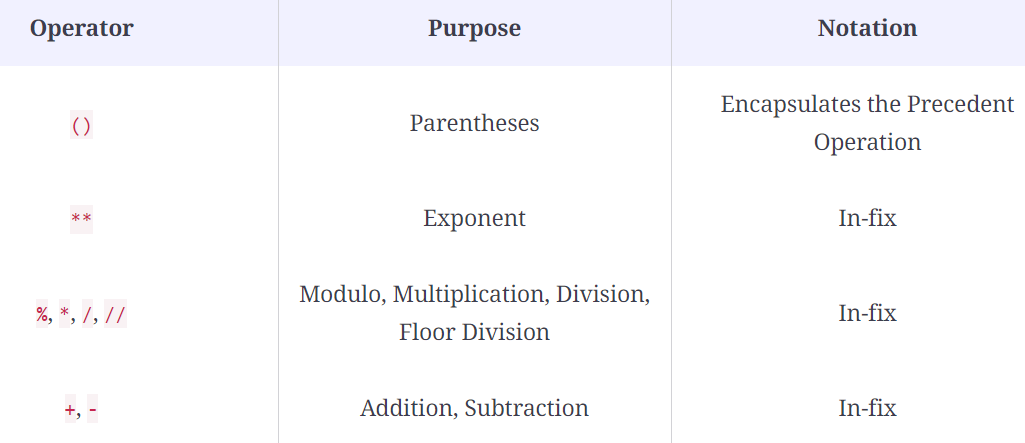
In this lesson, we'll learn how to perform calculations using arithmetic operators.

**We'll cover the following**

* [Addition](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gxxLA4WEoD3#Addition)
* [Subtraction](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gxxLA4WEoD3#Subtraction)
* [Multiplication](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gxxLA4WEoD3#Multiplication)
* [Division](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gxxLA4WEoD3#Division)
  + [Floor Division](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gxxLA4WEoD3#Floor-Division)
* [Modulo](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gxxLA4WEoD3#Modulo)
* [Precedence](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gxxLA4WEoD3#Precedence)
* [Parentheses](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gxxLA4WEoD3#Parentheses)

Below, we can find the basic arithmetic operators in order of **precedence**. The operator listed higher will be computed first.

These operators allow us to perform arithmetic operations in Python.



## Addition

We can add two numbers using the + operator:

print(10 + 5)

float1 = 13.65

float2 = 3.40

print(float1 + float2)

num = 20

flt = 10.5

print(num + flt)

As we can see in **line 9**, summing an integer and floating-point number gives us a floating-point number.

Python automatically converts the integer to a floating-point number. This applies to all arithmetic operations.

## Subtraction

We can subtract one number from the other using the - operator:

print(10 - 5)

float1 = -18.678

float2 = 3.55

print(float1 - float2)

num = 20

flt = 10.5

print(num - flt)

## Multiplication

We can multiply two numbers using the \* operator:

print(40 \* 10)

float1 = 5.5

float2 = 4.5

print(float1 \* float2)

print(10.2 \* 3)

## Division

We can divide one number by another using the / operator:

print(40 / 10)

float1 = 5.5

float2 = 4.5

print(float1 / float2)

print(12.4 / 2)

A division operation always results in a floating-point number.

### Floor Division

In floor division, the result is floored to the nearest smaller integer. It is also known as **integer division**.

For floor division, we must use the // operator:

print(43 // 10)

float1 = 5.5

float2 = 4.5

print(5.5 // 4.5)

print(12.4 // 2)

Unlike normal division, floor division between two integers results in an integer.

## Modulo

A number’s [modulo](https://en.wikipedia.org/wiki/Modulo_operation) with another number can be found using the % operator:

print(10 % 2)

twenty\_eight = 28

print(twenty\_eight % 10)

print(-28 % 10) # The remainder is positive if the right-hand operand is positive

print(28 % -10) # The remainder is negative if the right-hand operand is negative

print(34.4 % 2.5) # The remainder can be a float

## Precedence

An arithmetic [expression](https://en.wikipedia.org/wiki/Expression_(computer_science)) containing different operators will be computed on the basis of **operator precedence**.

Whenever operators have equal precedence, the expression is computed from the left side:

# Different precedence

print(10 - 3 \* 2) # Multiplication computed first, followed by subtraction

# Same precedence

print(3 \* 20 / 5) # Multiplication computed first, followed by division

print(3 / 20 \* 5) # Division computed first, followed by multiplication

## Parentheses

An expression which is enclosed inside parentheses will be computed first, regardless of operator precedence:

print((10 - 3) \* 2) # Subtraction occurs first

print((18 + 2) / (10 % 8))

Using all the operations above, we can compute complex mathematical expressions in Python!

Keep in mind that we are never restricted to these operators only, there are countless more arithmetic utilities at our disposal.

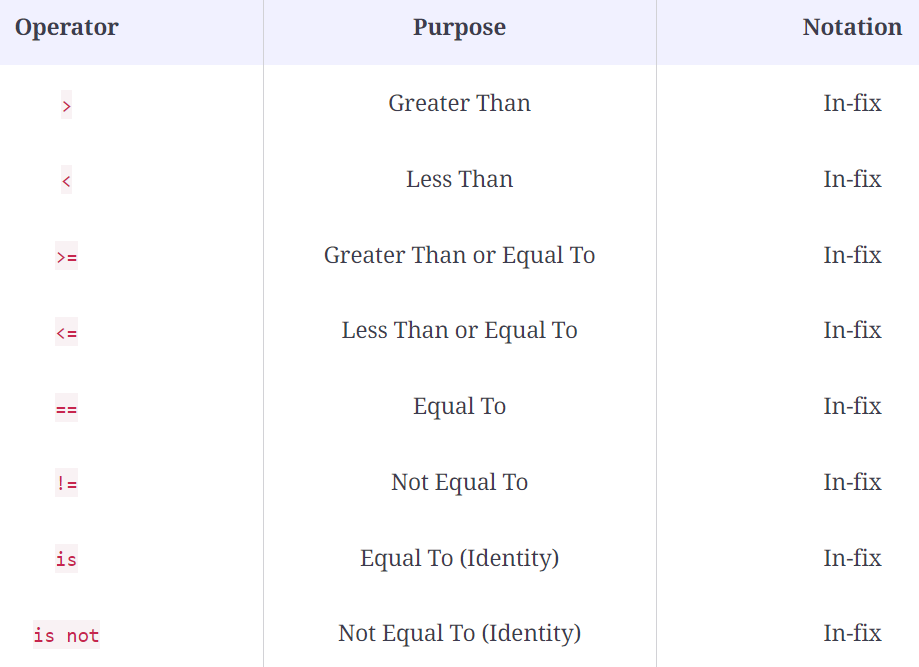
# Comparison Operators

In this lesson, we'll learn how to perform comparisons in Python using comparison operators.

**We'll cover the following**

* [Comparisons](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gx9GPY2Bk4Z#Comparisons)

Comparison operators can be used to compare values in mathematical terms.



## Comparisons

The result of a comparison is always a bool.

If the comparison is correct, the value of the bool will be True. Otherwise, its value will be False.

The == and != operators compare the **values** of both operands. However, the identity operators, is and is not, check whether the two operands are the **exact same object**.

Let’s look at a few examples:

num1 = 5

num2 = 10

num3 = 10

list1 = [6,7,8]

list2 = [6,7,8]

print(num2 > num1) # 10 is greater than 5

print(num1 > num2) # 5 is not greater than 10

print(num2 == num3) # Both have the same value

print(num3 != num1) # Both have different values

print(3 + 10 == 5 + 5) # Both are not equal

print(3 <= 2) # 3 is not less than or equal to 2

print(num2 is not num3) # Both have the same object

print(list1 is list2) # Both have the different objects

As we can see in **line 7**, num2 is indeed greater than num1. Hence, the result is True. On the other hand, **line 8** contains an incorrect comparison, which results in False.

# Assignment Operators

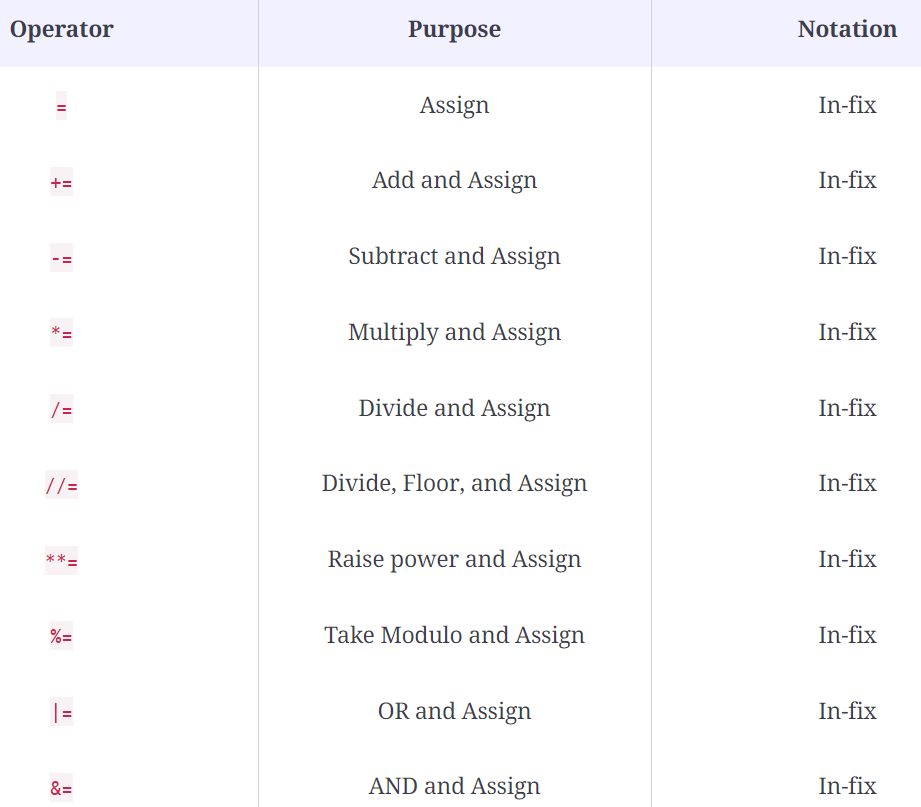
This lesson showcases Python's various assignment operators and their purpose.

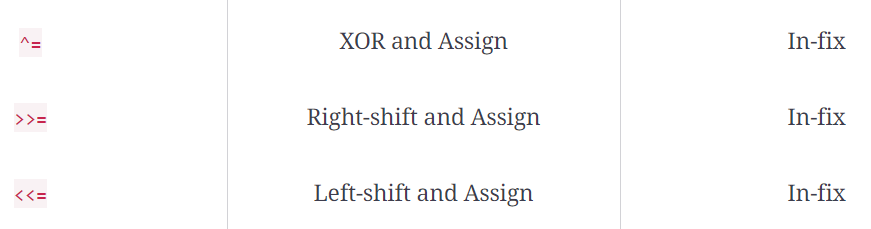
**We'll cover the following**

* [Assigning Values](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gxKAg37px63#Assigning-Values)
* [The Other Operators](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gxKAg37px63#The-Other-Operators)

This is a category of operators which is used to assign values to a variable. The = operator is an assignment operator, but not the only one.

Here’s a list of all the assignment operators supported in Python:





## Assigning Values

Let’s go through a few examples to see how values are assigned to variables.

Variables are **mutable**, so we can change their values whenever we want!

year = 2019

print(year)

year = 2020

print(year)

year = year + 1 # Using the existing value to create a new one

print(year)

One thing to note is that when a variable, first, is assigned to another variable, second, its value is **copied** into second. Hence, if we later change the value of first, second will remain unaffected:

first = 20

second = first

first = 35 # Updating 'first'

print(first, second) # 'second' remains unchanged

## The Other Operators

Below, we can see some of the assignment operators we talked about in action:

num = 10

print(num)

num += 5

print(num)

num -= 5

print(num)

num \*= 2

print(num)

num /= 2

print(num)

num \*\*= 2

print(num)

# Try all the others here!

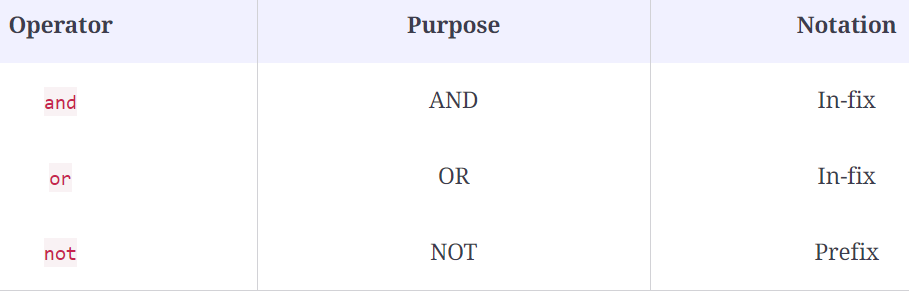
# Logical Operators

Let's understand the purpose of logical operators!

**We'll cover the following**

* [Logical Expressions](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/g2BV7nQ3o6D#Logical-Expressions)
* [Bit Value](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/g2BV7nQ3o6D#Bit-Value)

Logical operators are used to manipulate the logic of Boolean expressions.



## Logical Expressions

Logical expressions are formed using Booleans and logical operators.

Below, we can find some examples:

# OR Expression

my\_bool = True or False

print(my\_bool)

# AND Expression

my\_bool = True and False

print(my\_bool)

# NOT expression

my\_bool = False

print(not my\_bool)

## Bit Value

All the code we see around us in today’s world is actually made up of bits. Combinations of 1s and 0s form the foundation of programming.

In bit terms, the value of True is 1. False corresponds to 0:

print(10 \* True)

print(10 \* False)

The Python interpreter can automatically convert the bool to its numerical form when needed.

# Bitwise Operators

This lesson showcases all the different bitwise operators available in Python.

**We'll cover the following**

* [Examples](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/JYQZjDAgoJo#Examples)
* [Explanation](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/JYQZjDAgoJo#Explanation)

In programming, all data is actually made up of 0s and 1s known as bits. Bitwise operators allow us to perform bit-related operations on values.



num1 = 10 # Binary value = 01010

num2 = 20 # Binary Value = 10100

print(num1 & num2) # 0 -> Binary value = 00000

print(num1 | num2) # 30 -> Binary value = 11110

print(num1 ^ num2) # 30 -> Binary value = 11110

print(~num1) # -11 -> Binary value = -(1011)

print(num1 << 3) # 80 -> Binary value = 0101 0000

print(num2 >> 3) # 2 -> Binary value = 0010

**Explanation**

In **line 4**, we perform the bitwise AND. This operation takes a bit from num1 and the corresponding bit from num2 and performs an AND between them.

In simple terms, AND can be thought of as a multiplication between the two operands.

Now, let’s visualize this example:

* num1 is 01010 in binary and num2 is 10100.
* At the first step, the first binary digits of both numbers are taken:
  + **0**1010
  + **1**0100
* 0 & 1 would give 0 (again, think of it as multiplication).
* Next, we take the second digits:
  + 0**1**010
  + 1**0**100
* These two will once again give us 0.
* Doing this for all pairs, we can see that the answer is 0 each time.
* Hence, the output is 00000.

The OR operation in **line 5** will work in the same principle except that instead of multiplication, we will perform addition between the two binary numbers.

0 OR 1 gives us 1. 1 OR 1 also produces 1 (binary numbers do not go beyond 1). However, 0 OR 0 will give us 0 (0 + 0 is still 0).

Bitwise XOR and NOT will work on each bit as well. You can play with the code to get a better idea.

The bitshift operations (>> and <<) simply move the bits to either the right or the left. When a binary number is shifted, a 0 also enters at the opposite end to keep the number of the same size.

Let’s suppose we have a binary number 0110 (6 in decimal). The operation we perform is 0110 >> 2:

* 0**11**0 >> 2
* 00**11** (move one step to the right)
* 000**1** (move one more step to the right)
* Operation complete

Similarly, we can move 0110 twice to the left with the following operation 0110 << 2:

* 0**11**0 << 2
* 0**11**00 (move one step to the left)
* 0**11**000 (move one more step to the left)
* Operation complete

Note: In Python, leading zeroes are truncated. For example, the number **0011** will be the same as **11**. Similarly, the number **0001011** will be the same as **1011**.

# String Operations

This lesson showcases some of the most commonly used string operations.

**We'll cover the following**

* [Comparison Operators](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gkjp699GqWG#Comparison-Operators)
* [Concatenation](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gkjp699GqWG#Concatenation)
* [Search](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gkjp699GqWG#Search)

The string data type has numerous utilities that make string computations much easier. Let’s get down to the basics.

## Comparison Operators

Strings are compatible with the comparison operators. Each character has a [Unicode](https://en.wikipedia.org/wiki/Unicode) value.

This allows strings to be compared on the basis of their Unicode values.

When two strings have different lengths, the string which comes first in the dictionary is said to have the smaller value.

Let’s look at a few examples:

print('a' < 'b') # 'a' has a smaller Unicode value

house = "Gryffindor"

house\_copy = "Gryffindor"

print(house == house\_copy)

new\_house = "Slytherin"

print(house == new\_house)

print(new\_house <= house)

print(new\_house >= house)

## Concatenation

The + operator can be used to merge two strings together:

first\_half = "Bat"

second\_half = "man"

full\_name = first\_half + second\_half

print(full\_name)

The \* operator allows us to multiply a string, resulting in a repeating pattern:

print("ha" \* 3)

## Search

The in keyword can be used to check if a particular substring exists in another string. If the substring is found, the operation returns true.

Here’s how it works:

random\_string = "This is a random string"

print('of' in random\_string) # Check whether 'of' exists in randomString

print('random' in random\_string) # 'random' exists!

# Grouping Values

Now, we'll learn how to store multiple values together.

**We'll cover the following**

* [Making a List](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/YVGEZBBxRw0#Making-a-List)

In Python, we can store multiple values together in a single variable. While there are many ways of doing so, the most popular is the **list**.

It is very similar to a string since a string is a collection of characters. A list is also just a collection of values. However, the values can be of any type.

All we have to do is enclose all the elements in square brackets, [], and separate them with commas.

## Making a List [#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/YVGEZBBxRw0#Making-a-List)

my\_list = [1, 2.5, "A string", True]

print(my\_list)

It’s as simple as that! Lists can be indexed and sliced just like strings. The len command works with them too:

my\_list = [1, 2.5, "A string", True]

print(my\_list[2])

print(len(my\_list))

We’ll explore lists further as the course goes along. For now, we’re good to go ahead.

That brings us to the end of this section. By now, we should be familiar with the various data types, variables, and operators in Python.

In the next section, we’ll be introduced to **conditional statements**. Before that, be sure to check out our fun quiz and coding challenges on all the concepts we’ve learned so far!

What is the value of result at the end of the following code?

x = 20  
y = 5  
result = (x + True) / (4 - y \* False)

###### D)

5.25

What will be the output of the following piece of code?

my\_string = "0123456789"  
print(my\_string[-2: -6: -2])

86

# Exercise: Gravitational Force

Let's calculate the gravitational force between two masses!

**We'll cover the following**

* [Problem Statement](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/qALDB0zQYg2#Problem-Statement)
  + [Sample Input](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/qALDB0zQYg2#Sample-Input)
  + [Sample Output](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/qALDB0zQYg2#Sample-Output)
* [Coding Challenge](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/qALDB0zQYg2#Coding-Challenge)

## Problem Statement

Gravitational force is the attractive force that exists between two masses. It can be calculated by using the following formula:

����2*r*2*GMm*​

where G is the gravitational constant, M and m are the two masses, and r is the distance between them.

You must implement this equation in Python to calculate the gravitational force between Earth and the moon.

### Sample Input

G = 6.67 \* 10-11

MEarth = 6.0 \* 1024

mMoon = 7.34 \* 1022

r = 3.84 \* 108

### Sample Output

FG = 1.99 \* 1020

## Coding Challenge

All the values have already been given to you. You must write the formula in Pythonic syntax and store the answer in the grav\_force variable.

If you feel stuck, refer to the solution review in the next lesson.

Good luck!

G = 6.67 \* (10 \*\* -11)

M = 6.0 \* (10 \*\* 24) # Mass of Earth

m = 7.34 \* (10 \*\* 22) # Mass of the moon

r = 3.84 \* (10 \*\* 8)

# Write your code here

grav\_force = (G\*M\*m)/(r \*\* 2)

# What are Conditional Statements?

In this lesson, we will explore conditional statements and their functionality.

**We'll cover the following**

* [Definition](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/N77O3XBVqpv#Definition)
* [Conditional Statements in Python](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/N77O3XBVqpv#Conditional-Statements-in-Python)

## Definition

A conditional statement is a Boolean expression that, if True, executes a piece of code.

It allows programs to branch out into different paths based on Boolean expressions result in True or False outcomes.

In this way, conditional statements control the **flow** of the code and allow the computer to think. Hence, they are classified as **control structures**.

Conditional statements are an integral part of programming that every coder needs to know.

## Conditional Statements in Python

To handle conditional statements, Python follows a particular convention:

Press+to interact

if condtional statement is True:

*# execute expression1*

    pass

else:

*# execute expression2*

    pass

There are three types of conditional statements in Python:

* if
* if-else
* if-elif-else

# The if Statement

This lesson showcases the functionality of the 'if' statement.

**We'll cover the following**

* [The Structure](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/3YqV6xlwMq9#The-Structure)
  + [Indentation](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/3YqV6xlwMq9#Indentation)
  + [The Flow of an if Statement](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/3YqV6xlwMq9#The-Flow-of-an-if-Statement)
* [Conditions with Logical Operators](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/3YqV6xlwMq9#Conditions-with-Logical-Operators)
* [Nested if Statements](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/3YqV6xlwMq9#Nested-if-Statements)
* [Creating and Editing Values](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/3YqV6xlwMq9#Creating-and-Editing-Values)

## The Structure

The simplest conditional statement that we can write is the if statement. It comprises of two parts:

1. The **condition**
2. The **code to be executed**
3. The : in the illustration above is necessary to specify the beginning of the if statement’s code to be executed. However, the parentheses, (), around the condition are optional. The code to be executed is indented at least one tab to the right.

### Indentation

1. Indentation plays an essential role in Python. Statements with the same level of indentation belong to the same block of code. The code of an if statement is indented a space further than the code outside it in order to indicate that this is an inner and inter-related block.
2. The convention of our indents must also be consistent throughout a block. If we have used two spaces to make an indent, we must use two spaces for an indent in the same block. Hence, always keep indentation in mind when writing code.
3. Indents are important in other aspects of Python, too.

### The Flow of an if Statement

1. An if statement runs like this:
2. **if** the **condition** holds True, execute the **code to be executed**. Otherwise, **skip** it and move on.
3. Let’s write a simple if statement that verifies the value of an integer:
4. num = 5
5. if (num == 5): # The condition is true
6. print("The number is equal to 5") # The code is executed
7. if num > 5: # The condtion is false
8. print("The number is greater than 5") # The code is not executed
9. Our first condition simply checks whether the value of num is 5. Since this Boolean expression returns True, the compiler goes ahead and executes the print statement on **line 4**.
10. As we can see, the print command inside the body of the if statement is indented to the right. If it wasn’t, there would be an error. Python puts a lot of emphasis on proper indentation.

## Conditions with Logical Operators

1. We can use logical operators to create more complex conditions in the if statement. For example, we may want to satisfy multiple clauses for the expression to be True.
2. num = 12
3. if num % 2 == 0 and num % 3 == 0 and num % 4 == 0:
4. # Only works when num is a multiple of 2, 3, and 4
5. print("The number is a multiple of 2, 3, and 4")
6. if (num % 5 == 0 or num % 6 == 0):
7. # Only works when num is either a multiple of 5 or 6
8. print("The number is a multiple of 5 and/or 6")
9. In the first if statement, all the conditions have to be fulfilled since we’re using the and operator.
10. In the second if statement, the Boolean expression would be true if either or both of the clauses are satisfied because we are using the or operator.

## Nested if Statements

1. A cool feature of conditional statements is that we can nest them. This means that there could be an if statement inside another!
2. Hence, we can use nesting to make complex conditions in our program:
3. num = 63
4. if num >= 0 and num <= 100:
5. if num >= 50 and num <= 75:
6. if num >= 60 and num <= 70:
7. print("The number is in the 60-70 range")

## Creating and Editing Values

In a conditional statement, we can edit the values of our variables.

Furthermore, we can create new variables.

num = 10

if num > 5:

num = 20 # Assigning a new value to num

new\_num = num \* 5 # Creating a new value called newNum

# The if condition ends, but the changes made inside it remain

print(num)

print(new\_num)

The if statement is the foundation of conditional programming in Python.

# The if-else Statement

This lesson showcases the main properties of an 'if-else' statement.

**We'll cover the following**

* [Structure](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/xV9Ey1xEx9q#Structure)
* [Benefits of if-else](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/xV9Ey1xEx9q#Benefits-of-if-else)
* [Conditional Expressions](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/xV9Ey1xEx9q#Conditional-Expressions)

What if we want to execute a different set of operations in case an if condition turns out to be False?

That is where the if-else statement comes into the picture.

## Structure

The if-else statement looks something like this:

There’s nothing too tricky going on here. If the **condition** turns out to be False, the code after the else: keyword is executed.

Hence, we can now perform two different actions based on the condition’s value.

The else keyword will be on the same indentation level as the if keyword. Its body will be indented one tab to the right just like the if statement.

Here’s the if-else statement in action:

num = 60

if num <= 50:

print("The number is less than or equal to 50")

else:

print("The number is greater than 50")

## Benefits of if-else

The example above could also be written with two if conditions:

num = 60

if num <= 50:

print("The number is less than or equal to 50")

if num > 50:

print("The number is greater than 50")

However, for the second if, we have to specify the condition again. This can be tricky when dealing with complex conditions. The else statement automatically handles all the situations when the if fails.

Do keep in mind that the else statement cannot exist on its own. It is merely a counterpart of the if statement. It can still contain its own nested if or if-else statements. We’ll leave that as an exercise for you to try on your own.

## Conditional Expressions

Conditional expressions use the functionality of an if-else statement in a different way.

The expression returns an output based on the condition we provide. This output can be stored in a variable.

A conditional expression can be written in the following way:

output\_value1 if condition else output\_value2

If the if condition is fulfilled, the output would be output\_value1. Otherwise, it would be output\_value2.

Let’s refactor the previous if-else statement into a conditional expression:

num = 60

output = "The number is less than or equal to 50" \

if num <= 50 else "The number is greater than 50"

print(output)

Please note that the backslash \ in the above code is only a line continuation character that can be used to split a single line into multiple lines.

# The if-elif-else Statement

This lesson highlights the main properties of the `if-elif-else` statement.

**We'll cover the following**

* [Structure](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/JPqPZJEJ06y#Structure)
* [Multiple elif Statements](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/JPqPZJEJ06y#Multiple-elif-Statements)

The if-else statement handles two sides of the same condition: True and False. This works very well if we’re working with a problem that only has two outcomes.

However, in programming, it isn’t always a True or False scenario, and a problem can have multiple outcomes.

This is where the if-elif-else statement shines. It is the most comprehensive conditional statement because it allows us to create multiple conditions easily.

The elif stands for **else if**, indicating that if the previous condition fails, try this one.

## Structure

The if and else blocks will remain the same. The elif statement comes in between the two.

Let’s write an if-elif-else statement which checks the state of a traffic signal and generates the appropriate response:

light = "Red"

if light == "Green":

print("Go")

elif light == "Yellow":

print("Caution")

elif light == "Red":

print("Stop")

else:

print("Incorrect light signal")

Now, our conditional statement caters to **all** possible values of light.

Try changing the value and see how the response changes.

## Multiple elif Statements

This is the beauty of the if-elif-else statement. We can have as many elifs as we require, as long as they come between if and else.

**Note**: An if-elif statement can exist on its own without an else block at the end. However, an elif cannot exist without an if statement preceding it (which naturally makes sense).

Let’s write a piece of code that checks whether the value of an integer is in the range of 0-9 and prints the word in English:

num = 5

if num == 0:

print("Zero")

elif num == 1:

print("One")

elif num == 2:

print("Two")

elif num == 3:

print("Three")

elif num == 4:

print("Four")

elif num == 5:

print("Five")

elif num == 6:

print("Six")

elif num == 7:

print("Seven")

elif num == 8:

print("Eight")

elif num == 9:

print("Nine")

An important thing to keep in mind is that an if-elif-else or if-elif statement is not the same as multiple if statements. if statements act **independently**.

If the conditions of two successive ifs are True, both statements will be executed.

On the other hand, in if-elif-else, when a condition evaluates to True, the rest of the statement’s conditions are not evaluated.

We’ll understand this better through an example:

num = 10

if num > 5:

print("The number is greater than 5")

if num % 2 == 0:

print("The number is even")

if not num % 2 == 0:

print("The number is odd")

As we can see, in the if tab, all the statements are computed one by one. Hence, we get multiple outputs.

In the if-elif-else tab, since the first condition holds true, all the others are discarded. This proves to be more efficient in terms of code performance.

# Solution Review: Discounted Price

This review explains the solution for the 'Discounted Price' exercise.

**We'll cover the following**

* [Solution](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/B6kDx2zgQXo#Solution)
* [Explanation](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/B6kDx2zgQXo#Explanation)

## Solution [#](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/B6kDx2zgQXo#Solution)

price = 250

if price >= 300:

price \*= 0.7 # (1 - 0.3)

elif price >= 200:

price \*= 0.8 # (1 - 0.2)

elif price >= 100:

price \*= 0.9 # (1 - 0.1)

elif price < 100 and price >= 0:

price \*= 0.95 # (1 - 0.05)

print(price)

# What are Functions?

This lesson provides an in-depth look into the most powerful feature of Python programming: functions.

**We'll cover the following**

* [Definition](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/g2MBr9DkMY3#Definition)
  + [Why Use Functions?](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/g2MBr9DkMY3#Why-Use-Functions)
* [Types of Functions in Python](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/g2MBr9DkMY3#Types-of-Functions-in-Python)

## Definition

A function is a reusable set of operations.

That sounds like a pretty straightforward definition. But what does it exactly mean?

Take the print() and len() statements for instance. Both always perform predefined tasks. Hence, they are examples of functions!

### Why Use Functions?

Think of a function as a box which performs a task. We give it an input and it returns an output.

We don’t need to write the set of instructions again for a different input, we could just call the function again.

Functions are useful because they make the code concise and simple. The primary benefits of using functions are:

* **Reusability**: A function can be used over and over again. You do not have to write redundant code. For example, a sum() function could compute the sum of all the integers we provide it. We won’t have to write the summing operation ourselves each time.
* **Simplicity**: Functions are easy to use and make the code readable. We only need to know the inputs and the purpose of the function without focusing on the inner workings. This abstraction allows us to focus more on gaining the output instead of figuring out how it was computed.

An input isn’t even necessary. A function could perform its own computations to complete a task.

Suppose we want to find the smaller value between two integers:

num1 = 10

num2 = 40

if num1 < num2:

minimum = num1

else:

minimum = num2

print(minimum)

num1 = 250

num2 = 120

if num1 < num2:

minimum = num1

else:

minimum = num2

print(minimum)

num1 = 100

num2 = 100

if num1 < num2:

minimum = num1

else:

minimum = num2

print(minimum)

For every new pair of integers, we need to write the if-else statement again.

All this could become much simpler if we had a function to perform the steps necessary for calculating the minimum.

The good news is that Python already has the min() function:

minimum = min(10, 40)

print(minimum)

minimum = min(10, 100, 1, 1000) # It even works with multiple arguments

print(minimum)

minimum = min("Superman", "Batman") # And with different data types

print(minimum)

Now the code looks pretty awesome. Plus, it’s easier to write. And that is the beauty of functions!

**Types of Functions in Python**

Functions are perhaps the most commonly used feature of Python. There are two basic types of functions in Python:

1. Built-in functions
2. User-defined functions

len(), min(), and print() are examples of built-in functions.

The coolest feature, however, is that the language allows us to create our own functions that perform the tasks we require.

# Function Creation

In this lesson, we'll learn how to create a function.

**We'll cover the following**

* [Components of a Function](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/mEXYj1y5x03#Components-of-a-Function)
* [Implementation](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/mEXYj1y5x03#Implementation)
* [Function Parameters](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/mEXYj1y5x03#Function-Parameters)
* [The return Statement](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/mEXYj1y5x03#The-return-Statement)

## Components of a Function

How do we actually make a function? In Python, a function can be defined using the def keyword in the following format:

The function name is simply the name we’ll use to identify the function.

The parameters of a function are the inputs for that function. We can use these inputs within the function. Parameters are optional.

The body of the function contains the set of operations that the function will perform. This is always indented to the right.

## Implementation

Let’s start by making a plain function that prints four lines of text. It won’t have any parameters. We’ll name it my\_print\_function. We can call the function in our code using its name along with empty parentheses:

def my\_print\_function(): # No parameters

print("This")

print("is")

print("A")

print("function")

# Function ended

# Calling the function in the program multiple times

my\_print\_function()

my\_print\_function()

And just like that, we’ve created our first function! We can reuse it whenever we want.

## Function Parameters

**Parameters** are a crucial part of the function structure.

They are the means of passing data to the function. This data can be used by the function to perform a meaningful task.

When creating a function, we must define the number of parameters and their names. These names are only relevant to the function and won’t affect variable names elsewhere in the code. Parameters are enclosed in parentheses and separated by commas.

The actual values/variables passed into the parameters are known as **arguments**.

The min() function requires two numbers as inputs and prints the smaller one.

Let’s define our own basic form of the min() function that simply prints the minimum. We’ll name it minimum():

def minimum(first, second):

if (first < second):

print(first)

else:

print(second)

num1 = 10

num2 = 20

minimum(num1, num2)

Here, we are passing num1 and num2 to the function. The positions of the parameters are important. In the case above, the value of num1 will be assigned to first as it was the first parameter. Similarly, the value of num2 assigned to second.

If we call a function with lesser or more arguments than originally required, Python will throw an error.

A parameter can be any sort of data object; from a simple integer to a huge list.

## The return Statement

So far, we’ve only defined functions that print something. They don’t return anything back to us. But if we think back, functions return values all the time. Just take len() for example. It returns an integer which is the length of the data structure.

To return something from a function, we must use the return keyword. Keep in mind that once the return statement is executed, the compiler ends the function. Any remaining lines of code after the return statement will not be executed.

Let’s refactor the minimum() method to return the smaller value instead of printing it. Now, it’ll work just like the built-in min() function with two parameters, if we can repair the following code to make it run successfully. Our AI Mentor can guide us as well.

def minimum(first, second):

if (first < second):

return first

return second

num1 = 10

num2 = 20

result = minimum(num1,num2) # Storing the value returned by the function

print(result)

We’ve learned how to create a function, set its parameters, provide it arguments, and return data from it. Python really does make it a simple process.

It is a good practice to define all our functions first and then begin the main code. Defining them first ensures that they can be used anywhere in the program safely.

**Function Scope**

This lesson will highlight some important things to know about a function's scope.

**We'll cover the following**

* [Data Lifecycle](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/JYmYgLl0Mxg#Data-Lifecycle)
* [Altering Data](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/JYmYgLl0Mxg#Altering-Data)

The scope of a function means the extent to which the variables and other data items made inside the function are accessible in code.

In Python, the function scope is the function’s body.

Whenever a function runs, the program moves into the function scope. It moves back to the outer scope once the function has ended.

**Data Lifecycle**

In Python, data created inside the function cannot be used from the outside unless it is being returned from the function.

Variables in a function are isolated from the rest of the program. When the function ends, they are released from memory and cannot be recovered.

The following code will never work:

def func():

name = "Stark"

func()

print(name) # Accessing 'name' outside the function

As we can see, the name variable doesn’t exist in the outer scope, and Python lets us know.

Similarly, the function cannot access data outside its scope unless the data has been passed in as an argument.

name = "Ned"

def func():

name = "Stark"

func()

print(name) # The value of 'name' remains unchanged.

**Altering Data**

When **mutable** data is passed to a function, the function can modify or alter it. These modifications will stay in effect outside the function scope as well. An example of mutable data is a list.

In the case of **immutable** data, the function can modify it, but the data will remain unchanged outside the function’s scope. Examples of immutable data are numbers, strings, etc.

Let’s try to change the value of an integer inside a function:

num = 20

def multiply\_by\_10(n):

n \*= 10

num = n # Changing the value inside the function

print("Value of num inside function:", num)

return n

multiply\_by\_10(num)

print("Value of num outside function:", num) # The original value remains unchanged

So, it’s confirmed that immutable objects are unaffected by the working of a function. If we really need to update immutable variables through a function, we can simply assign the returning value from the function to the variable.

Now, we’ll try updating a mutable object through a function:

num\_list = [10, 20, 30, 40]

print(num\_list)

def multiply\_by\_10(my\_list):

my\_list[0] \*= 10

my\_list[1] \*= 10

my\_list[2] \*= 10

my\_list[3] \*= 10

multiply\_by\_10(num\_list)

print(num\_list) # The contents of the list have been change

We passed num\_list to our function as the my\_list parameter. Now, any changes made to my\_list will reflect in num\_list outside the function. This would not happen in the case of an immutable variable.

By this point, we have a better understanding of the scope of a function.

**Built-In String Functions**

In this lesson, we'll take a look at some of the built-in functions offered by Python.

**We'll cover the following**

* [Strings](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/Y5nMnQMg4DO#Strings)
  + [Search](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/Y5nMnQMg4DO#Search)
  + [Replace](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/Y5nMnQMg4DO#Replace)
  + [Changing the Letter Case](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/Y5nMnQMg4DO#Changing-the-Letter-Case)
  + [Joining strings](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/Y5nMnQMg4DO#Joining-strings)
  + [Formatting](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/Y5nMnQMg4DO#Formatting)

Python boasts a huge library of built-in functions.

And trust us when we say that there’s something for almost everyone.

**Strings**

Functions that are properties of a particular entity are known as **methods**. These methods can be accessed using the . operator. The string data type has several methods associated with it. Let’s look at some of them.

**Search**

An alternative for finding a substring using the in keyword is the find() method. It returns the first index at which a substring occurs in a string. If no instance of the substring is found, the method returns -1.

-1 is a conventional value that represents a None or failure in case the output was supposed to be positive.

For a string called a\_string, find() can be used in the following way:

a\_string.find(substring, start, end)

* substring is what we are searching for.
* start is the index from which we start searching in a\_string.
* end is the index where we stop our search in a\_string.

start and end are optional.

random\_string = "This is a string"

print(random\_string.find("is")) # First instance of 'is' occurs at index 2

print(random\_string.find("is", 9, 13)) # No instance of 'is' in this range

**Replace**

The replace() method can be used to replace a part of a string with another string. Here’s the template we must use:

a\_string.replace(substring\_to\_be\_replaced, new\_string)

The original string is not altered. Instead, a new string with the replaced substring is returned.

a\_string = "Welcome to Educative!"

new\_string = a\_string.replace("Welcome to", "Greetings from")

print(a\_string)

print(new\_string)

**Changing the Letter Case**

In Python, the letter case of a string can be easily changed using the upper() and lower() methods.

Let’s try them out:

print("UpperCase".upper())

print("LowerCase".lower())

**Joining strings**

With join() method you can join multiple strings. Let’s try it out:

llist = ['a', 'b', 'c']

print('>>'.join(llist)) # joining strings with >>

print('<<'.join(llist)) # joining strings with <<

print(', '.join(llist)) # joining strings with comma and space

Over here, the string.join(llist) returns a single string, with the elements of the llist separated by string.

**Formatting**

The format() method can be used to format the specified value(s) and insert them in string’s placeholder(s). Let’s try it out:

string1 = "Learn Python {version} at {cname}".format(version = 3, cname = "Educative")

string2 = "Learn Python {0} at {1}".format(3, "Educative")

string3 = "Learn Python {} at {}".format(3, "Educative")

print(string1)

print(string2)

print(string3)

The placeholders can be identified using named indexes {cname}, numbered indexes {0}, or even empty placeholders {}.

**Lambdas**

In this lesson, we'll study a unique category of functions called lambda.

**We'll cover the following**

* [Definition](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/q2By3pxYOAk#Definition)
* [Syntax](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/q2By3pxYOAk#Syntax)
* [The Purpose of Lambdas](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/q2By3pxYOAk#The-Purpose-of-Lambdas)

We have to specify function names while creating them. However, there is a special class of functions for which we do not need to specify function names.

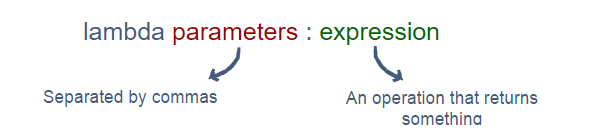
**Definition**

A **lambda** is an anonymous function that *returns* some form of data.

Lambdas are defined using the lambda keyword. Since they return data, it is a good practice to assign them to a variable.

**Syntax**

The following syntax is used for creating lambdas:



In the structure above, the parameters are optional.

Let’s try creating a few simple lambdas.

Below, we can find a lambda that triples the value of the parameter and returns this new value:

triple = lambda num : num \* 3 # Assigning the lambda to a variable

print(triple(10)) # Calling the lambda and giving it a parameter

Here’s a simple lambda function that aims to concatenate the first characters of three strings together, if we can repair the following code to make it run successfully. Our AI Mentor can guide us as well.

concat\_strings = lambda a, b, c: a[0] + b[0]

print(concat\_strings("World", "Wide", "Web"))

As we can see, lambdas are simpler and more readable than normal functions. But this simplicity comes with a limitation.

A lambda cannot have a multi-line expression. This means that our expression needs to be something that can be written in a single line.

Hence, lambdas are perfect for short, single-line functions.

We can also use conditional statements within lambdas:

my\_func = lambda num: "High" if num > 50 else "Low"

print(my\_func(60))

When using conditional statements in lambdas, the if-else pair is necessary. Both cases need to be covered, otherwise, the lambda will throw an error:

my\_func = lambda num: "High" if num > 50

**The Purpose of Lambdas**

So, what is the point of having lambdas around? We’re still assigning them to variables, so they do have names.

They can be written in-line, but that isn’t a huge advantage.

Well, lambdas are really useful when a function requires another function as its argument.

**Functions as Arguments**

This lesson teaches us how to use functions as parameters for other functions.

**We'll cover the following**

* [Using Simple Functions](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/qAWxKNNWYYD#Using-Simple-Functions)
* [Using Lambdas](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/qAWxKNNWYYD#Using-Lambdas)
* [More Examples](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/qAWxKNNWYYD#More-Examples)

In Python, one function can become an argument for another function. This is useful in many cases.

Let’s make a calculator function that requires the add, subtract, multiply, or divide function along with two numbers as arguments.

For this, we’ll have to define the four arithmetic functions as well.

**Using Simple Functions**

def add(n1, n2):

return n1 + n2

def subtract(n1, n2):

return n1 - n2

def multiply(n1, n2):

return n1 \* n2

def divide(n1, n2):

return n1 / n2

def calculator(operation, n1, n2):

return operation(n1, n2) # Using the 'operation' argument as a function

result = calculator(multiply, 10, 20)

print(result)

print(calculator(add, 10, 20))

Python automatically understands that the multiply argument in **line 21** is a function, and so, everything works perfectly.

**Using Lambdas**

For the calculator method, we needed to write four extra functions that could be used as the argument. This can be quite a hassle.

Why don’t we just pass a lambda as the argument? The four operations are pretty simple, so they can be written as lambdas.

Let’s try it:

def calculator(operation, n1, n2):

return operation(n1, n2) # Using the 'operation' argument as a function

# 10 and 20 are the arguments.

result = calculator(lambda n1, n2: n1 \* n2, 10, 20)

# The lambda multiplies them.

print(result)

print(calculator(lambda n1, n2: n1 + n2, 10, 20))

The code looks much shorter now! We can define the operation on the go whenever we want.

This is the beauty of lambdas. They work really well as arguments for other functions.

**More Examples**

The built-in map() function creates a **map object** using an existing list and a function as its parameters. This object can be converted to a list using the list() function.

The template for map() is as follows:

map(function, list)

The function will be applied, or *mapped*, to all the elements of the list.

Below, we’ll use map() to double the values of an existing list:

num\_list = [0, 1, 2, 3, 4, 5]

double\_list = map(lambda n: n \* 2, num\_list)

print(list(double\_list))

This creates a new list. The original list remains unchanged.

We could have created a function that doubles a number and used it as the argument in map(), but the lambda made things simpler.

Another similar example is the filter() function. It requires a function and a list.

filter() *filters* elements from a list if the elements satisfy the condition that is specified in the argument function.

Let’s write a filter() function that filters all the elements which are greater than 10:

numList = [30, 2, -15, 17, 9, 100]

greater\_than\_10 = list(filter(lambda n: n > 10, numList))

print(greater\_than\_10)

The function returns a **filter object** which can be converted to a list using list().

just like map(), filter() returns a new object without changing the original list.

By now, we have a better understanding of how functions can become arguments and why lambdas are helpful in that situation.

**Recursion**

This lesson will explain the concept of recursion in Python.

**We'll cover the following**

* [Definition](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gkGRgw9G91j#Definition)
* [A Simple Example](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gkGRgw9G91j#A-Simple-Example)
* [Why Use Recursion?](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gkGRgw9G91j#Why-Use-Recursion)
* [A Complex Example](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/gkGRgw9G91j#A-Complex-Example)

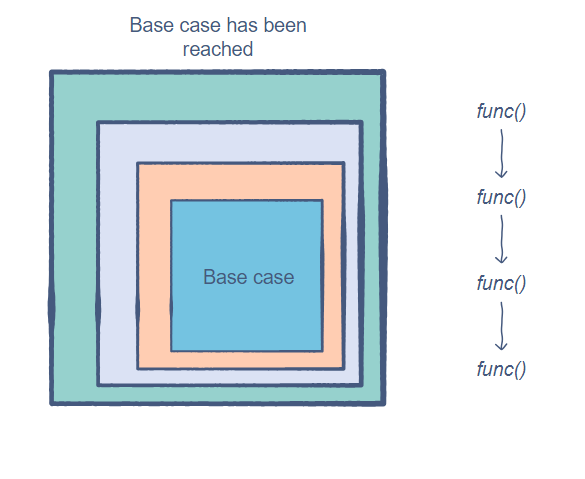
**Definition**

Recursion is the process in which a function calls itself during its execution. Each recursive call takes the program one scope deeper into the function.

The recursive calls stop at the **base case**. The base case is a check used to indicate that there should be no further recursion.

Imagine recursive calls as nested boxes where each box represents a function call. Each call makes a new box. When the base case is reached, we start moving out of the boxes one by one:

As we can see in the illustration above, the function is called repeatedly for a specified number of times.



**A Simple Example**

Let’s write a function which decrements a number recursively until the number becomes 0:

def rec\_count(number):

print(number)

# Base case

if number == 0:

return 0

rec\_count(number - 1) # A recursive call with a different argument

print(number)

rec\_count(5)

3 2 1 0 1 2 3 4 5

This is fairly easy to understand. In each call, the value of the number variable is printed. We then check whether the base case has been fulfilled. If not, we make a recursive call to the function with the current value decremented.

One thing to notice is that an outer call cannot move forward until all the inner recursive calls have finished. This is why we get a sequence of 5 to 0 to 5.

**Why Use Recursion?**

Recursion is a concept which many find difficult to grasp at first, but it has its advantages. For starters, it can significantly reduce the runtime of certain algorithms, which makes the code more efficient.

Recursion also allows us to easily solve many problems related to **graphs** and **trees**. It is also important in search algorithms.

However, we need to be careful when using recursion. If we don’t specify an appropriate base case or update our arguments as we recurse, the program will reach **infinite recursion** and crash. The arguments passed to our recursive function are updated in each recursive call so that the base case can eventually be reached.

**A Complex Example**

The Fibonacci sequence is a popular series of numbers in mathematics, where every number is the sum of the two numbers before it. The first two terms in the series are 0 and 1:

0 1 1 2 3 5 8 13

Let’s write a function which takes in a number, n, and returns the **nth** number in the Fibonacci sequence. It is important to note that for the following example, we will be treating all inputs **less than** 1 as incorrect and therefore, our input will start from 1. So, if n == 6, the function will return 5:

Press+to interact

Ace Editor

def fib(n):

# The base cases

if n <= 1: # First number in the sequence

return 0

elif n == 2: # Second number in the sequence

return 1

else:

# Recursive call

return fib(n - 1) + fib(n - 2)

print(fib(6))

Output

5

First, we handle our base cases. We know that the first two values are always 0 and 1, so that is where we can stop our recursive calls.

If n is larger than 2, then it will be the sum of the two values before it.

**Solution Review: Repetition and Concatenation**

This review explains the solution for the 'Repetition and Concatenation' exercise.

**We'll cover the following**

* [Solution](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/YVV0nJj8pGn#Solution)
* [Explanation](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/YVV0nJj8pGn#Explanation)

**Solution**

def rep\_cat(x, y):

return str(x) \* 10 + str(y) \* 5

print(rep\_cat(3, 4))

**Explanation**

To convert the integers into strings, we can use the str() method.

The \* operator is perfect for replicating the value of a string a certain number of times.

Two strings can easily be concatenated using the + operator.

Finally, all of this is returned from the function using the return statement.

**Solution Review: The Factorial!**

def factorial(n):

# Base case

if n == 0 or n == 1:

return 1

if n < 0:

return -1

# Recursive call

return n \* factorial(n - 1)

print(factorial(5))

**Explanation**

This problem can easily be solved using recursion. The base case is when n is 1 or 0 since it’s the minimum we can go. In either case, we return 1, since it is the factorial for both these values.

Other than that, the only special case is if n is negative. That can be handled with a simple if statement.

The final and most important step is the recursive call. Each call returns a product back to the previous call where the product is multiplied with the current value of n in that particular call.

**What are Loops?**

This lesson explains the functionality of loops in Python.

**We'll cover the following**

* [Definition](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/RLW5gAOV9lO#Definition)
* [Loops in Python](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/RLW5gAOV9lO#Loops-in-Python)

**Definition**

A loop is a **control structure** that is used to perform a set of instructions for a specific number of times.

Loops solve the problem of having to write the same set of instructions over and over again. We can specify the number of times we want the code to execute.

One of the biggest applications of loops is traversing data structures, e.g. lists, tuples, sets, etc. In such a case, the loop **iterates** over the elements of the data structure while performing a set of operations each time.

Just like conditional statements, a loop is classified as a control structure because it directs the flow of a program by making varying decisions in its **iterations**.

Loops are a crucial part of many popular programming languages such as C++, Java, and JavaScript.

**Loops in Python**

There are two types of loops that we can use in Python:

1. The for loop
2. The while loop

Both differ slightly in terms of functionality.

**The for Loop**

This lesson highlights the main features of the 'for' loop.

**We'll cover the following**

* [Structure](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/3wkYJPOo8jp#Structure)
* [Looping Through a Range](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/3wkYJPOo8jp#Looping-Through-a-Range)
* [Looping Through a List/String](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/3wkYJPOo8jp#Looping-Through-a-ListString)

A for loop uses an **iterator** to traverse a sequence, e.g. a range of numbers, the elements of a list, etc. In simple terms, the iterator is a variable that goes through the list.

The iterator starts from the beginning of the sequence. In each iteration, the iterator updates to the next value in the sequence.

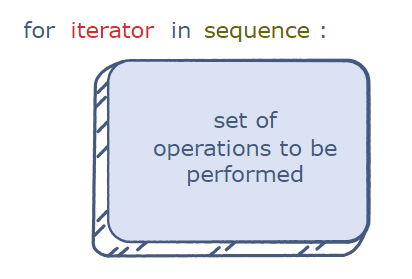
The loop ends when the iterator reaches the end.

**Structure**

In a for loop, we need to define three main things:

1. The name of the iterator
2. The sequence to be traversed
3. The set of operations to perform

The loop always begins with the for keyword. The body of the loop is indented to the right:



The in keyword specifies that the iterator will go through the values *in* the sequence/data structure.

**Looping Through a Range**

In Python, the built-in range() function can be used to create a sequence of integers. This sequence can be iterated over through a loop. A range is specified in the following format:

range(start, end, step)

The end value is not included in the list.

If the start index is not specified, its default value is 0.

The step decides the number of steps the iterator jumps ahead after each iteration. It is optional and if we don’t specify it, the default step is 1, which means that the iterator will move forward by one step after each iteration.

Let’s take a look at how a for loop iterates through a range of integers:

for i in range(1, 11): # A sequence from 1 to 10

if i % 2 == 0:

print(i, " is even")

else:

print(i, " is odd")

As we can see above, rather than individually checking whether every integer from 1 to 10 is even or odd, we can loop through the sequence and compute i % 2 == 0 for each element.

The iterator, i, begins from 1 and becomes every succeeding value in the sequence.

Let’s see how a loop changes when the step component of a range is specified:

for i in range(1, 11, 3):

print(i)

**Looping Through a List/String**

A list or string can be iterated through its indices.

Let’s double each value in a list using a for loop:

float\_list = [2.5, 16.42, 10.77, 8.3, 34.21]

print(float\_list)

for i in range(0, len(float\_list)): # Iterator traverses to the last index of the list

float\_list[i] = float\_list[i] \* 2

print(float\_list)

We could also traverse the elements of a list/string directly through the iterator. In the float\_list above, let’s check how many elements are greater than 10:

float\_list = [2.5, 16.42, 10.77, 8.3, 34.21]

count\_greater = 0

for num in float\_list: # Iterator traverses to the last index of the list

if num > 10:

count\_greater += 1

print(count\_greater)

In this example, num is the iterator. An important thing to keep in mind is that in the case above, altering num will **not alter the actual value in the list**. The iterator makes a copy of the list element.

**Nested for Loops**

In this lesson, we'll create nested 'for' loops!

**We'll cover the following**

* [Execution of Nested Loops](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/qVZYY02mgR3#Execution-of-Nested-Loops)
* [Using a Nested for Loop](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/qVZYY02mgR3#Using-a-Nested-for-Loop)
* [The break Keyword](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/qVZYY02mgR3#The-break-Keyword)
* [The continue Keyword](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/qVZYY02mgR3#The-continue-Keyword)
* [The pass Keyword](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/qVZYY02mgR3#The-pass-Keyword)

**Execution of Nested Loops**

Python lets us easily create loops within loops. There’s only one catch: the inner loop will always complete before the outer loop.

For each iteration of the outer loop, the iterator in the inner loop will complete its iterations for the given range, after which the outer loop can move to the next iteration.

**Using a Nested for Loop**

Let’s take an example. Suppose we want to print two elements whose sum is equal to a certain number n.

The simplest way would be to compare every element with the rest of the list. A nested for loop is perfect for this:

Press+to interact

Ace Editor

n = 50

num\_list = [10, 25, 4, 23, 6, 18, 27, 47]

for n1 in num\_list:

for n2 in num\_list: # Now we have two iterators

if(n1 != n2):

if(n1 + n2 == n):

print(n1, n2)

n the code above, each element is compared with every other element to check if n1 + n2 is equal to n. This is the power of nested loops!

**The break Keyword**

Sometimes, we need to exit the loop before it reaches the end. This can happen if we have found what we were looking for and don’t need to make any more computations in the loop.

A perfect example is the one we have just covered. At a certain point, n1 is 23 and n2 is 27. Our condition of n1 + n2 == n has been fulfilled. But the loops keep running and comparing all other pairs as well. This is why the pair is printed twice. It would be nice to just stop it when the pair is found once.

That’s what the break keyword is for. It can *break* the loop whenever we want.

Let’s add it to the example above:

n = 50

num\_list = [10, 25, 4, 23, 6, 18, 27, 47]

found = False # This bool will become true once a pair is found

for n1 in num\_list:

for n2 in num\_list:

if(n1 != n2):

if(n1 + n2 == n):

found = True # Set found to True

break # Break inner loop if a pair is found

if found:

print(n1, n2) # Print the pair

break # Break outer loop if a pair is found

As we can see, only (23, 27) is printed this time.

This is because (23, 27) is the first pair which satisfies the condition. We terminate the loop after that using the found bool. Hence, (27, 23) is never computed.

**The continue Keyword**

When the continue keyword is used, the rest of that particular iteration is skipped. The loop *continues* on to the next iteration. We can say that it doesn’t break the loop, but it skips all the code in the current iteration and moves to the next one.

We don’t need to get into too much detail, so here’s a simple example:

num\_list = list(range(0, 10))

for num in num\_list:

if num == 3 or num == 6 or num == 8:

continue

print(num)

The loop goes into the if block when num is 3, 6, or 8. When this happens, continue is executed and the rest of the iteration, including the print() statement, is skipped.

**The pass Keyword**

In all practical meaning, the pass statement does nothing to the code execution. It can be used to represent an area of code that needs to be written. Hence, it is simply there to assist you when you haven’t written a piece of code but still need your entire program to execute.

num\_list = list(range(20))

for num in num\_list:

pass # You can write code here later on

print(len(num\_list))

**The while Loop**

This lesson highlights the key features of the 'while' loop.

**We'll cover the following**

* [Structure](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/R14plX52XVK#Structure)
* [The while Loop in Action](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/R14plX52XVK#The-while-Loop-in-Action)
* [Cautionary Measures](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/R14plX52XVK#Cautionary-Measures)
* [Other Properties](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/R14plX52XVK#Other-Properties)
* [Iteration vs. Recursion](https://www.educative.io/module/lesson/python-fundamentals-for-programmers/R14plX52XVK#Iteration-vs-Recursion)

The while loop keeps iterating over a certain set of operations as long as a certain **condition** holds True.

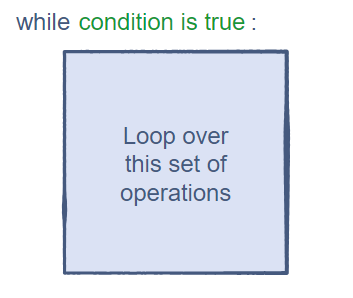
It operates using the following logic:

*While this condition is true, keep the loop running.*

**Structure**

In a for loop, the number of iterations is fixed since we know the size of the sequence.

On the other hand, a while loop is not always restricted to a fixed range. Its execution is based solely on the condition associated with it.



**The while Loop in Action**

Here’s a while loop that finds out the maximum power of n before the value exceeds 1000:

n = 2 # Could be any number

power = 0

val = n

while val < 1000:

power += 1

val \*= n

print(power) In each iteration, we update val and check if its value is less than 1000. The value of power tells us the maximum power n can have before it becomes greater than or equal to 1000. Think of it as a counter.

We can also use while loops with data structures, especially in cases where the length of data structure changes during iterations.

The following loop computes the sum of the first and the last digits of any integer:

n = 249

last = n % 10 # Finding the last number is easy

first = n # Set it to `n` initially

while first >= 10:

first //= 10 # Keep dividing by 10 until the leftmost digit is reached.

result = first + last

print(result)

**Cautionary Measures**

Compared to for loops, we should be more careful when creating while loops. This is because a while loop has the potential to never end. This could crash a program!

Have a look at these simple loops:

while(True):

print("Hello World")

x = 1

while(x > 0):

x += 5

The loops above will never end because their conditions always remain true. Hence, we should always make sure that our condition has a mutable variable/object that is being updated in the loop and will eventually turn the condition false.

**Other Properties**

The break, continue, and pass keywords work with while loops.

Like for loops, we can also nest while loops. Furthermore, we can nest the two types of loops with each other.

**Iteration vs. Recursion**

If we observe closely, there are several similarities between iteration and recursion. In recursion, a function performs the same set of operations repeatedly but with different arguments.

A loop does the same thing except that the value of the iterator and other variables in the loop’s body change in each iteration.

Figuring out which approach to use is an intuitive process. Many problems can be solved through both.

Recursion is useful when we need to divide data into different chunks. Iteration is useful for traversing data and also when we don’t want the program’s scope to change.

**Solution Review: Balanced Brackets**

This review explains the solution for the 'Balanced Brackets' exercise.

def check\_balance(brackets):

check = 0

for bracket in brackets:

if bracket == '[':

check += 1

elif bracket == ']':

check -= 1

if check < 0:

break

return check == 0

bracket\_string = '[[]]'

print(check\_balance(bracket\_string))

**Explanation**

The solution relies on the value of the check variable, which is updated in each iteration. If an opening bracket is found, check is incremented by 1. In the case of a closing bracket, check is decremented by 1.

The logic is that check should **never** be negative because that would imply that somewhere in the string, there are more closing brackets than opening ones. The condition for being unbalanced is satisfied and we don’t need to check further.

Another case is that after the loop finishes, the value of check would be 0 because the brackets match. If it’s not, the function simply returns False.

**Solution Review: A Sum of Zero**

This review explains the solution for the 'A Sum of Zero' exercise.

def check\_sum(num\_list):

for first\_num in range(len(num\_list)):

for second\_num in range(first\_num + 1, len(num\_list)):

if num\_list[first\_num] + num\_list[second\_num] == 0:

return True

return False

num\_list = [10, -14, 26, 5, -3, 13, -5]

print(check\_sum(num\_list))

**Explanation**

We can use a nested for loop to obtain all the possible pairs in the list.

By using range(), we can make sure that the inner loop always starts one index ahead of the outer loop.

This removes the possibility of a comparison being repeated.

**Solution Review: Fibonacci Series**

This review explains the solution for the 'Fibonacci Series' problem.

def fib(n):

# The first and second values will always be fixed

first = 0

second = 1

if n < 1:

return -1

if n == 1:

return first

if n == 2:

return second

count = 3 # Starting from 3 because we already know the first two values

while count <= n:

fib\_n = first + second

first = second

second = fib\_n

count += 1 # Increment count in each iteration

return fib\_n

n = 7

print(fib(n))